Efficient classical simulation of the Deutsch-Jozsa and Simon's algorithms

Niklas Johansson and Jan-Åke Larsson
Institutionen för systemteknik, Linköpings Universitet, 581 83 Linköping, SWEDEN

(Dated: Nov 19, 2015)

A long-standing aim of quantum information research is to understand what gives quantum computers their advantage. This requires separating problems that need genuinely quantum resources from those for which classical resources are enough. Two examples of quantum speed-up are the Deutsch-Jozsa and Simon's problem, both efficiently solvable on a quantum Turing machine, and both believed to lack efficient classical solutions. Here we present a framework that can simulate both quantum algorithms efficiently, solving the Deutsch-Jozsa problem with probability 1 using only one oracle query, and Simon's problem using linearly many oracle queries, just as expected of an ideal quantum computer. The presented simulation framework is in turn efficiently simulatable in a classical probabilistic Turing machine. This shows that the Deutsch-Jozsa and Simon's problem do not require any genuinely quantum resources, and that the quantum algorithms show no speed-up when compared with their corresponding classical simulation. Finally, this gives insight into what properties are needed in the two algorithms, and calls for further study of oracle separation between quantum and classical computation.

Quantum computational speed-up has motivated much research to build quantum computers, to find new algorithms, to quantify the speed-up, and to separate classical from quantum computation. One important goal is to understand the reason for quantum computational speed-up; to understand what resources are needed to do quantum computation. Some candidates for such necessary resources include superposition and interference [1], entanglement [2], nonlocality [3], contextuality [4–6], and the continuity of state-space [7]. In this paper we look at two so-called *oracle* problems: the Deutsch-Jozsa problem [8, 9] and Simon's problem [10, 11], thought to show that quantum computation is more powerful than classical computation. There are quantum algorithms that solve these problems efficiently, and here we present a framework that can simulate these quantum algorithms, *Quantum Simulation Logic* (QSL), that in itself can be efficiently simulated on a classical probabilistic Turing machine. To underline this, we provide Python implementations of these simulations as supplementary material, that give the correct output in seconds for 100,000-qubit inputs to the Deutsch-Jozsa algorithm and 512-qubit inputs to Simon's algorithm, for random choice of the function under consideration. This shows that no genuinely quantum resources are needed to solve these problems efficiently, but also tells us which properties are actually needed: the possibility to choose between two aspects of the same system within which to store, process, and retrieve information.

The quantum algorithm for the Deutsch-Josza problem has been used extensively for illustrating experimental realizations of a quantum computer, while Simon's algorithm served as inspiration for the later Shor's algorithm [13]. Both problems involve finding certain properties of a function f, and this can be done efficiently in a quantum computer given a quantum gate that implements the function, known as an *oracle*. When using a classical function as an oracle (an oracle that is only allowed to act on classical bits as input and giving classical bits as output), the Deutsch-Jozsa problem has an efficient solution on a classical probabilistic Turing machine [8, 9], but for Simon's problem it has been proven [10, 11] that no efficient solution exists. In both the quantum and classical case the oracle is treated as a black box, i.e., its internal structure is not visible from the outside, and each oracle call counts as one operation; one unit

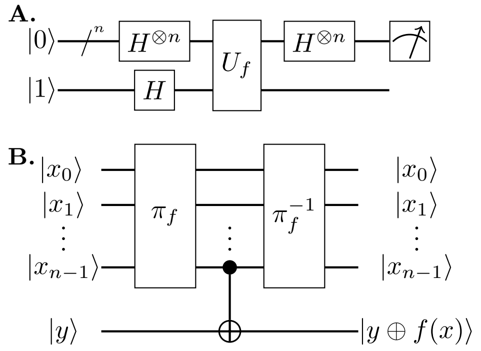

FIG. 1: Circuits for the Deutsch-Jozsa algorithm. **A.** The algorithm itself. This circuit uses an n-qubit query-register prepared in the state $|0\rangle$, and a target prepared in $|1\rangle$. It proceeds to apply Hadamard transformations to each qubit. The function f is embedded in an oracle U_f, and this is followed by another Hadamard transformation on each query-register qubit. The measurement at the end will test positive for $|0\rangle$ if f was constant, and negative if f was balanced. **B.** Oracle construction for balanced functions. The arbitrary permutation $\pi = \sum_x |\pi(x)\rangle\langle x|$ of the computational basis states is constructed from CNOT and Toffoli gates [12] ($\pi^{-1} = \pi^\dagger$). At the center is a CNOT gate from the most significant qubit $|x_n\rangle$ to the target $|y\rangle$. With π as the identity permutation, the oracle performs the balanced function f' that is 1 for all inputs with the most significant bit set. Any other balanced function $f(x) = f'(\pi(x))$ can now be generated by choosing a different π. For a function that is constant zero, the CNOT gate is omitted, while the constant one function should replace the CNOT with a Pauli-X gate acting on the target.

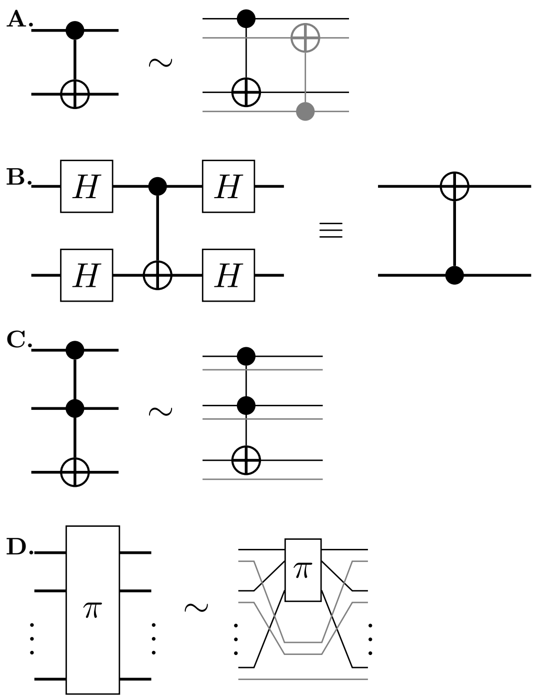

FIG. 2: QSL circuitry. **A.** QSL construction of a CNOT gate (computational bit in black, phase bit in grey). **B.** CNOT gate identity valid both for qubit and QSL gates. **C.** QSL construction of a Toffoli gate. **D.** QSL construction of a general permutation.

of time. An algorithm that requires at most a polynomial number of oracle calls and at most a polynomial amount of additional resources is regarded as efficient, but the oracle itself does not need to be efficiently computable. In essence, the question asked is "If this oracle was efficiently computable, would the whole algorithm be efficiently computable?" For both the mentioned problems, the answer is "yes" in quantum computation. For Simon's problem, the answer is "no" for the classical-function oracle, giving evidence for a separation *relative to the oracle* between quantum computation and classical computation. For the Deutsch-Jozsa problem such a separation is only obtained by requiring a deterministic solution.

I. THE DEUTSCH-JOZSA ALGORITHM AND QUANTUM SIMULATION LOGIC

The Deutsch-Jozsa problem is the following: Suppose that you are given a Boolean function $f(x) : \{0,1\}^n \mapsto \{0,1\}$ with the promise that it is either constant or balanced. The function is constant if it gives the same output (1 or 0) for all possible inputs, and it is balanced if it gives the output 0 for half of the possible inputs, and 1 for the other half. Your task is now to distinguish between these two cases [9, 14]. Given such a function, a classical Turing machine can solve this problem by checking the output for $2^{n-1}+1$ values of the input; if all are the same, the function is constant, and otherwise balanced. A stochastic algorithm with k randomized function queries gives a bounded error probability [9] less than 2^{1-k}, showing that the problem is in the complexity class **BPP** (Bounded-error Probabilistic Polynomial-time solvable problems).

A quantum computer obtains the function as a unitary that implements the function, a quantum oracle. For later convenience we use the gate implementation of the oracle U_f shown in Figure 1b. This device will give the expected outcomes for regular function queries, i.e., invocations that reveal a single function value. The desired input number k should be inserted in the query-register as the quantum state $|k\rangle$ in the computational basis, and the answer-register should be in a computational basis state (say $|0\rangle$). Applying the oracle and measuring the answer-register will reveal the output of f: if the answer-register was flipped, then the function value is 1 for that input k. Given this quantum oracle the Deutsch-Jozsa algorithm [9, 14] can solve the problem with a single query (see Figure 1a). In this algorithm Hadamards are applied to the query-register state, creating a quantum superposition of all possible inputs to the oracle. When the oracle is applied, this superposition is unchanged if the function is constant, and changed to a different superposition if the function is balanced. A change can be detected by again applying Hadamards and measuring in the computational basis. In the ideal case the error probability is zero, showing that the problem is in **EQP** (Exact or Error-free Quantum Polynomial-time solvable problems [15]).

The QSL versions of the quantum algorithm will use the same set-up as in Figure 1, but instead of qubits it will use pairs (b_z, b_x) containing two classical bits, a "computational" bit b_z and a "phase" bit b_x. These constitute the elementary systems of our QSL framework, the "QSL bits". State preparation of a computational single qubit state $|k\rangle$ is associated with preparation of (k, X) where X is a random evenly distributed bit. Measurement in the computational basis is associated with readout of the computational bit followed by randomization of the phase bit, somewhat like the uncertainty relation or really measurement disturbance as seen in quantum mechanics. Accordingly, measurement of the phase bit will be followed by a randomization of the computational bit. These constructions of state preparation and measurement prohibits exact preparation and readout of the system; the upper limit is one bit of information per QSL bit (b_z, b_x).

It is relatively simple to simulate the single qubit gates used in the quantum algorithms: an X gate inverts the computational bit (preserving the value of the phase bit b_x), a Z gate inverts the phase bit (preserving b_z), and a H gate switches the computational and phase bits. These are constructed so that the resulting gates obey the quantum identities $XX = ZZ = HH = I$ and $HZH = X$. It is also possible to define QSL Y and "phase" gates, but we will not discuss these and their associated identities here as they are not needed for the present task. A simple circuit is to prepare the state $|0\rangle$, apply an H gate, and measure in the computational basis. This will give a random evenly distributed bit as output.

A system of several qubits can now be associated with a system of several QSL bits containing two internal bits each. A QSL simulation of the quantum Controlled-NOT (CNOT) gate can be constructed from two classical reversible-logic CNOTs. One classical CNOT connects the computational bits in the "forward" direction, while the other connects the phase bits in the "reverse" direction, see Figure 2a. This again is constructed to enable use of the same identities as apply for the quantum CNOT, most importantly a phenomenon sometimes called "phase kick-back" in quantum computation [14], see Figure 2b.

Our QSL framework is inspired by a construction of Spekkens [16] that captures many, but not all, properties of quantum mechanics. So far, the presented framework is completely equivalent to Spekkens' model (the H gate appears in a paper by Pusey [17]). Both frameworks are efficiently simulatable on a classical Turing machine (see above and Ref. [16]). Furthermore, both have a close relation to stabilizer states and Clifford group operations [17], and perhaps more interestingly, both frameworks capture some properties of entanglement, enabling protocols like super-dense coding and quantum-like teleportation, but cannot give all consequences of entanglement, most importantly, cannot give a Bell inequality violation.

Returning to the Deutsch-Josza problem, for $n = 1$ the oracle for the balanced function $f(x) = x$ is a CNOT, which means that Figure 2b contains the whole Deutsch-Josza algorithm when using the prescribed initial state $|0\rangle|1\rangle$. With this initial state, the query-register measurement result would be 1, so that this oracle gives the same output from the QSL algorithm as from the quantum algorithm. Indeed, all oracles for $n = 1$ and $n = 2$ only use CNOT and X gates, so their simulation will give the same behaviour as the quantum oracles. For $n = 1$ and $n = 2$, the Deutsch-Jozsa algorithm is already known to have an implementation that does not rely on quantum resources [18], and also, the gates used so far can be found within the stabilizer formalism, and here the Gottesmann-Knill theorem tells us that they can be simulated efficiently [19].

However, for $n \geq 3$, the Deutsch-Jozsa oracle needs the Toffoli gate, and since the Toffoli gate cannot be efficiently simulated using the stabilizer formalism [19], nor is present in Spekkens' framework [16, 20], it has so far been believed that the Deutsch-Jozsa algorithm does not have an efficient classical simulation. Here, we need to point out that our task is not to create exact Toffoli gate equivalents, or even simulate the full quantum-mechanical system as such. It suffices to give a working efficient QSL version of the Deutsch-Jozsa algorithm. We therefore choose not to represent Toffoli gates exactly, but design the gate so that it implements a classical Toffoli in the computational bits, and some other gate in the phase bits. For simplicity we choose the identity map for the phase bits, see Figure 2c.

By representing a general computational-state permutation π in the same manner as the Toffoli (see Figure 2d), the quantum oracle can immediately be translated into an oracle within the QSL framework. The balanced-function oracle in Figure 1b leaves the phase bits unchanged except for the one belonging to the most significant QSL bit in the query-register. A constant-function oracle leaves all phase bits unchanged. Since Hadamard gates are included before and after the oracle, measurement of the computational bits will reveal the oracle's effect on the phase bits of the query-register, solving the problem by only one oracle query.

This QSL algorithm uses equally many QSL bits and QSL gates as the quantum algorithm uses qubits and quantum gates. The time and space complexity are therefore identical to the complexity of the quantum algorithm. The QSL algorithm can be efficiently simulated on a classical probabilistic Turing machine, using two classical bits for each simulated qubit and at most twice as many classical reversible gates as quantum gates. The error probability is 0, and in fact, the same correct output is generated for all possible random values in the probabilistic machine, the machine's random value could therefore be replaced with a constant value without destroying the simulation. Therefore, this QSL algorithm can be simulated on a classical *deterministic* Turing machine, showing that, relative to the oracle, the Deutsch-Jozsa problem is in **P** (Polynomial time solvable problems).

II. SIMON'S PROBLEM

Simon's problem is to decide whether a function $f(x) : \{0,1\}^n \mapsto \{0,1\}^n$ is one-to-one or two-to-one invariant under the bitwise exclusive-OR operation with a secret string s (or equivalent, invariant under a nontrivial XOR-mask s [10, 11]). In both cases, $f(x) = f(x')$ if and only if $x' = x \oplus s$ (bit-wise addition modulo 2); if $s = 0$ then f is one-to-one and otherwise f is two-to-one. When using a classical function as an oracle, a classical probabilistic Turing machine needs an exponential number of oracle evaluations for this task [10, 11], showing that relative to this oracle, Simon's problem is not in **BPP**.

Simon's quantum algorithm can distinguish these two cases in an expected linear number of oracle queries [10, 11], showing that the problem is in **BQP** (Bounded-error Quantum Polynomial time [15]). A circuit diagram of the quantum subroutine and one realization of the oracle used in Simon's algorithm is shown in Figure 3a–b. An explicit construction of the gate U_s can be obtained by choosing a basis $\{v^k\}$ for the part of the binary vector space orthogonal to s. If $s = 0$, the basis consists of n vectors, otherwise $n - 1$ vectors. For every entry $v^k_j = 1$, connect a CNOT from query-register bit j to ancilla-

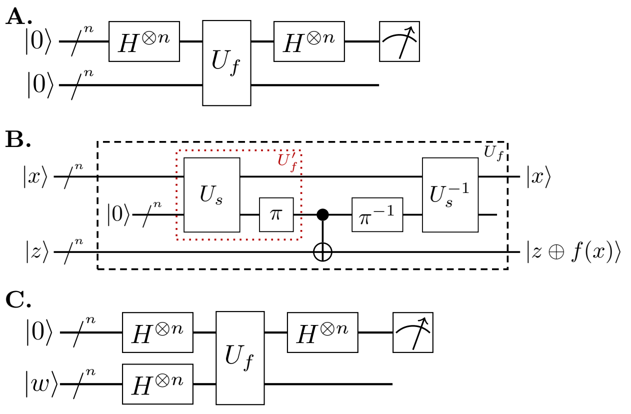

FIG. 3: Circuits for the quantum subroutine of Simon's algorithm. **A.** The subroutine itself. The n-qubit input and output registers are prepared in the state $|0\rangle|0\rangle$. Apply Hadamard transformations to the query-register, the function f embedded in an oracle U_f, and finally another Hadamard. The measurement at the end will give a random bit sequence y such that $y \cdot s = 0 \pmod 2$. **B.** Oracle construction. The oracle U_f uses an internal n-qubit ancilla, and the CNOT denotes a string of bit-wise CNOTs from each ancilla bit to the corresponding answer-register bit. The unitary U_s is constructed from CNOTs [21] (see the main text) so that it implements one representative function for each specific secret string s, and the permutation π gives access to all functions f. The dotted gate combination U_f' implements $|x\rangle|0\rangle \mapsto |x\rangle|f(x)\rangle$ as required in Simon [10, 11], and the additional circuitry in U_f is there to map $|x\rangle|z\rangle$ to $|x\rangle|z \oplus f(x)\rangle$ and reset the ancilla [22]. **C.** The modified subroutine for the deterministic algorithm. The n-qubit query- and answer-register are prepared in the state $|0\rangle|w\rangle$. The circuit applies Hadamard transformations to the query- *and answer-registers*, the function f embedded in an oracle U_f, and finally another Hadamard before measurement.

register k (in realizations, it is beneficial to choose v^k to minimize the number of CNOTs [21]). The permutation after U_s enables all possible functions.

Let us illustrate the construction of U_s with the following example. Suppose $s = (1, 0, 1)$, then we choose the basis $\{v^k\} = \{(1, 0, 1), (0, 1, 0)\}$ which spans the binary vector space orthogonal to s (note that the scalar product is defined mod 2). Now we see that $U_{(1,0,1)}$ can be constructed from three CNOTs, one from the first query-register qubit to the first ancilla qubit, one from the third query-register qubit to the first ancilla qubit, and one from the second query-register qubit to the second ancilla qubit.

The quantum subroutine output is a random bit vector y such that $y \cdot s = 0$ mod 2 (bit-wise dot product), uniformly distributed over all such bit vectors y. After an expected number of iterations that is linear in n, the subroutine will have produced $n-1$ linearly independent values of y [10, 11], so that the resulting linear system of equations for s can be solved, giving one non-trivial solution s^*. If the function is one-to-one, this solution is just a random bit sequence, but if the function is two-to-one the solution is the actual secret string s. Querying the function for $f(0)$ and $f(s^*)$ will give equal values if the function is two-to-one, and unequal values if the function is one-to-one, solving Simon's problem. Note that as a consequence of solving Simon's problem one also obtains the secret string s.

The QSL versions of the subroutine and oracle is again obtained by substituting the quantum gates with their corresponding QSL gates. The QSL CNOT and U_s constructions give the explicit map $(x,p),(0,a) \mapsto \big(x, p \oplus g'(a)\big), \big(f'(x), a\big)$ where

$$\begin{aligned} f_j'(x) &= x \cdot v^j \pmod 2 = f_j'(x \oplus s), \\ g_j'(a) &= \sum_k a_k v_j^k \pmod 2. \end{aligned} \tag{1}$$

The permutations π and π^{-1} do not influence the phase, so the complete oracle U_f is the map $(x,p),(z,w) \mapsto \big(x, p \oplus g(w)\big), \big(z \oplus f(x), w\big)$ where

$$\begin{aligned} f(x) &= \pi(f'(x)) = f(x \oplus s), \\ g(w) &= g'(a) \oplus g'(w \oplus a) = g'(w) \end{aligned} \tag{2}$$

By the use of Hadamard gates, the simulation of Simon's subroutine sets the query-register phase entering the oracle to $p = 0^n$, while the answer-register phase w will be uniformly distributed. Measurement of the computational bits after the final Hadamard will therefore give a random bit vector $y = g(w)$ that is orthogonal to s, uniformly distributed over the possible values. This reproduces the quantum predictions exactly, showing that the simulation will work with this subroutine to solve Simon's problem with the same expected linear number of iterations in n. Again, efficient simulation of the QSL framework on a classical probabilistic Turing machine shows that, relative to the oracle, Simon's problem is in **BPP**.

The derandomized quantum algorithm for Simon's problem [23, 24] (with zero error probability), that shows that Simon's problem relative to the quantum oracle is in **EQP**, is not immediately usable because it incorporates a modified Grover search [25] which is not known to have an implementation within the QSL framework. The quantum algorithm is intended to remove already seen y and, crucially, prevent measuring $y = 0$ which would be a failed iteration of the algorithm. The modified Grover search is presented [23] as applied to the output of U_f, but since the final step of the Grover search is again U_f, the whole combination can be viewed as U_f preceded by another quantum algorithm. This other quantum algorithm, in a sense, finds an input to U_f such that the coefficient of the term $|0\rangle$ of the output is 0.

Since we do not have a Grover equivalent in our framework, we would like a simpler solution. One way to achieve this is to prepare the initial answer-register state as the Hadamard transformation of a chosen state $|w\rangle$, see Figure 3c. Iterating $|w\rangle$ over a basis for the bit-vector space will produce n values of y that span the bit vector space orthogonal to s. Either this spans the whole space ($s = 0$), or gives a linear system of equations that has one non-trivial solution, equal to s. This QSL algorithm gives deterministic correct outputs just as the QSL Deutsch-Jozsa algorithm, and can therefore also be efficiently simulated on a classical deterministic Turing machine, showing that relative to the oracle, Simon's problem actually lies within **P**.

This linear search procedure works in our setting because the QSL permutation gate π does not influence the phase bits. In the quantum-mechanical case, the quantum gate implementing π does change what is sometimes called *phase information*, e.g., in the simple case when the quantum π is a CNOT. A more convoluted example that does not correspond to a simple modification of the phase information is when the quantum π is a (non-stabilizer) Toffoli gate. Thus, the derandomized quantum algorithm [23] requires the more advanced modified Grover's search algorithm. However, note that if the oracle is the smaller gate combination U_f' in Figure 3b that obeys only the original [10, 11] requirement $|x\rangle|0\rangle \mapsto |x\rangle|f(x)\rangle$ (for $z = 0$), then the quantum version of the simpler algorithm proposed above will work as well, avoiding the modified Grover search.

III. CONCLUSIONS

In conclusion, we have devised QSL equivalents of the Deutsch-Jozsa and Simon's quantum algorithms. In the quantum algorithm, you are given a quantum oracle, a unitary gate that implements f by acting on qubits, your task is to distinguish two or more families of functions. In the presented QSL algorithms, you are given a gate that implements f by acting on QSL systems, and the same task. Using the QSL framework, we obtain the same success probability and the same time and space complexity as for the quantum algorithms, and these QSL algorithms are in turn efficiently simulatable in classical Turing machines (implementations that can be run with very large inputs are provided as Supplementary material). Therefore, the Deutsch-Jozsa and Simon's algorithms cannot anymore be used as evidence for a relativized oracle separation between **EQP** and **BPP**, or even **P**. Both problems are, relative to the oracles, in fact in **P**.

Apparently, the resource that gives these quantum algorithms their power is also available in QSL, but not when using an oracle in the form of an ordinary classical function. Each qubit is simulated by two classical bits in a classical probabilistic Turing machine, and in the above oracles one of these bits is used for computing the function, while the other is changed systematically when U_f is applied. This change (in *phase information*) can then be revealed by choosing to insert Hadamards before and after the oracle. This choice is present in both quantum systems and our simulation, and enables revealing either function output or function structure, as desired. Our conclusion is that the needed resource is the possibility to choose between two aspects of the same system within which to store, process, and retrieve information.

While we still believe that quantum computers *are* more powerful than classical computers, the question arises whether oracle separation really can distinguish quantum-computational complexity classes from classical complexity classes. There are many other examples of quantum-classical oracle separation, [26–28] and some simple cases can be conjectured to have efficient simulations in the QSL framework (e.g., 3-level systems [16] for the three-valued Deutsch-Jozsa problem [26]), but in general the question needs further study. It *is* possible that the same technique can be used for direct computation using other quantum algorithms [13, 25], but this will take us out of the oracle paradigm. Also, general quantum computation has been conjectured to use genuinely quantum properties such as the continuum of quantum states, or contextuality [5, 6], which both are missing from the QSL framework [16]. In any case, neither the Deutsch-Jozsa nor Simon's algorithm needs genuinely quantum mechanical resources.

Acknowledgments

The authors would like to thank Peter Jonsson, Chris Fuchs, and David Mermin for discussions on the subject. The project is supported by the Foundational Questions Institute (FQXi) through the Silicon Valley Community Foundation.

Supplementary Information

Available in the form of Python implementations of the QSL algorithms, supplied with the online version of the paper.

[1] R. P. Feynman, "Simulating physics with computers", International Journal of Theoretical Physics **21**, 467–488 (1982).

[2] A. Einstein, B. Podolsky, and N. Rosen, "Can Quantum-Mechanical Description of Physical Reality be Considered Complete?", Phys. Rev. **47**, 777–780 (1935).

[3] J. S. Bell, "On the Einstein-Podolsky-Rosen paradox", Physics (Long Island City, N. Y.) **1**, 195–200 (1964).

[4] S. Kochen and E. P. Specker, "The problem of hidden variables in quantum mechanics", Journal of Mathematics and Mechanics **17**, 59–87 (1967).

[5] J.-Å. Larsson, "A contextual extension of spekkens' toy model", in AIP conference proceedings, Vol. 1424 (2012), pp. 211–220.

[6] M. Howard, J. Wallman, V. Veitch, and J. Emerson, “Contextuality supplies the magic for quantum computation”, Nature **510**, 351–355 (2014).

[7] P. Shor, “Fault-tolerant quantum computation”, in Proceedings of 37th annual symposium on foundations of computer science (Oct. 1996), pp. 56–65.

[8] D. Deutsch, “Quantum theory, the Church-Turing principle and the universal quantum computer”, Proc. Roy. Soc. London A **400**, 97–117 (1985).

[9] D. Deutsch and R. Jozsa, “Rapid solution of problems by quantum computation”, Proc. Roy. Soc. Lond. A **439**, 553–558 (1992).

[10] D. Simon, “On the power of quantum computation”, in Proceedings of the 35th annual symposium on foundations of computer science (Nov. 1994), pp. 116–123.

[11] D. Simon, “On the power of quantum computation”, SIAM J. Comput. **26**, 1474–1483 (1997).

[12] M. A. Nielsen and I. L. Chuang, *Quantum computation and quantum information*, 10th Anniversary Ed. (Cambridge University Press, New York, USA, 2011).

[13] P. W. Shor, “Algorithms for Quantum Computation: Discrete Logarithms and Factoring”, in Proceedings of the 35th Annual Symposium on Foundations of Computer Science (Nov. 20, 1994), pp. 124–134.

[14] R. Cleve, A. Ekert, C. Macchiavello, and M. Mosca, “Quantum algorithms revisited”, Proc. Roy. Soc. London A **454**, 339–354 (1998).

[15] E. Bernstein and U. Vazirani, “Quantum complexity theory”, SIAM J. Comput. **26**, 1411–1473 (1997).

[16] R. W. Spekkens, “Evidence for the epistemic view of quantum states: a toy theory”, Phys. Rev. A **75**, 032110 (2007).

[17] M. F. Pusey, “Stabilizer notation for spekkens’ toy theory”, Found. Phys. **42**, 688–708 (2012).

[18] D. Collins, K. W. Kim, and W. C. Holton, “Deutsch-Jozsa algorithm as a test of quantum computation”, Phys. Rev. A **58**, R1633–R1636 (1998).

[19] D. Gottesman, “The heisenberg representation of quantum computers”, (1998).

[20] N. Johansson, *Efficient simulation of Deutsch-Jozsa algorithm* (Linköping University, The Institute of Technology, 2015).

[21] M. S. Tame, B. A. Bell, C. Di Franco, W. J. Wadsworth, and J. G. Rarity, “Experimental realization of a one-way quantum computer algorithm solving simon’s problem”, Physical Review Letters **113**, 200501 (2014).

[22] C. Bennett, “Logical reversibility of computation”, IBM Journal of Research and Development **17**, 525–532 (1973).

[23] G. Brassard and P. Hoyer, “An exact quantum polynomial-time algorithm for Simon’s problem”, in Theory of Computing and Systems, 1997., Proceedings of the Fifth Israeli Symposium on (1997), pp. 12–23.

[24] T. Mihara and S. C. Sung, “Deterministic polynomial-time quantum algorithms for Simon’s problem”, Computational Complexity **12**, 162–175 (2003).

[25] L. K. Grover, “A fast quantum mechanical algorithm for database search”, in Proceedings of the twenty-eighth annual ACM symposium on theory of computing, STOC ’96 (1996), pp. 212–219.

[26] Y. Fan, “A generalization of the deutsch-jozsa algorithm to multi-valued quantum logic”, in IEEE 44th international symposium on multiple-valued logic (2007), p. 12.

[27] G. Alagic, C. Moore, and A. Russell, “Quantum algorithms for simon’s problem over general groups”, in Proceedings of the eighteenth annual ACM-SIAM symposium on discrete algorithms, SODA ’07 (2007), pp. 1217–1224.

[28] *Oracular quantum algorithms, quantum algorithm zoo*, `http://math.nist.gov/quantum/zoo/#oracular` (visited on 08/14/2015).

Appendix A: Python library for the Quantum Simulation Logic

```
#! /usr/bin/env python

# Quantum_Simulation_Logic.py
# Version 1.0
# Copyright Jan-Ake Larsson <jan-ake.larsson@liu.se> 2015

import random
from math import log

class QSL_system(list):
  def __init__(self,value,n=None,phase=None):
    # QSL states are either initialized from bit strings or integers, and
    # preferrably the length n should be fixed if an integer is given
    list.__init__(self,[0,0])
    self.n=n
    if type(value)==str or type(value)==BinaryVector:
      self[0]=int(value,2)
      if not self.n:
        self.n=len(value)
    else:
      self[0]=value
      if not self.n:
        self.n=int(log(value*1.0,2))+1 # at least 1 bit
    self[1]=phase
    if not self[1]:
      self[1]=random.randint(0,(1<<self.n)-1)
  def __add__(self,other):
    new_comp=(self[0]<<other.n)+other[0]
    new_phase=(self[1]<<other.n)+other[1]
    new_n=self.n+other.n
    return QSL_system(new_comp,new_n,new_phase)
  def __str__(self):
    comp=bin(self[0])[2:]
    phase=bin(self[1])[2:]
    return "["+"0"*(self.n-len(comp))+comp+", "\
      +"0"*(self.n-len(phase))+phase+"]"
  def extend(self,other):
    self[0]=(self[0]<<other.n)+other[0]
    self[1]=(self[1]<<other.n)+other[1]
    self.n=self.n+other.n
  def remove(self,n):
    self[0]/=1<<n
    self[1]/=1<<n
    self.n-=n


def X(state,k):
  # k is the bit number to apply the gate on
  state[0]=state[0]^(1<<k)
def Z(state,k):
  # k is the bit number to apply the gate on
  state[1]=state[1]^(1<<k)
def Y(state,k):
  # k is the bit number to apply the gate on
  state[0]=state[0]^(1<<k)
  state[1]=state[1]^(1<<k)
def H(state,k):
  # If equal do nothing, but if not equal, flip both
  if state[0]&(1<<k) != state[1]&(1<<k):
    Y(state,k)
def CN(state,k):
  # k is the tuple (control bit number, target bit number)
  # Computational bit
  if state[0]&(1<<k[0]): # If control computational bit is 1
    state[0]=state[0]^(1<<k[1]) # Flip target computational bit
  # Phase bit
  if state[1]&(1<<k[1]): # If target phase bit is 1
    state[1]=state[1]^(1<<k[0]) # Flip control phase bit

# Measurement output is rudimentary binary vectors
class BinaryVector(str):
  # Immutable parent class: replace __new__, not __init__
  def __new__(self,value,n=None):
    if type(value)!=str:
      retval=bin(value)[2:]
    if n!=None:
      value="0"*(n-len(value))+value
    return str.__new__(self,value)
  def __add__(self,other):
    retval=BinaryVector(bin(int(self,2)^int(other,2))[2:],
                        max(len(self),len(other)))
    return retval
  def __radd__(self,other):
    return self+other
  def dot(self,other):
    return bin(int(self,2)&int(other,2)).count("1")%2

def measureZ(state,k=None):
  # Measure in the computational basis
  if k==None:
    k=xrange(state.n)
  retval=""
  for j in reversed(k): # Most significant bit first in string
    state[1]=state[1]^(random.randint(0,1)<<j)
    if state[0]&(1<<j):
      retval+="1"
    else:
      retval+="0"
  return BinaryVector(retval)

# Efficient representation of a random permutation _and_ its inverse.
class RandPerm(dict):
  # This would be a "reversible random oracle" in computer science: if the
  # input has not been seen before, draw a random output. If the input has
  # been seen, repeat the corresponding output. Make sure the random oracle is
  # one-to-one, and provide the inverse.
  def __init__(self, n, parent=None):
    dict.__init__(self)
    self.N=2**n-1
    if parent==None:
      self.inverse=RandPerm(n,self)
    else:
      self.inverse=parent
  def __getitem__(self, key):
    if (type(key)!=int and type(key)!=long) or key<0 or key>self.N:
      raise ValueError
    try:
      retval = dict.__getitem__(self, key)
    except:
      # Draw new items randomly, when needed.
      retval=random.randint(0,self.N)
      while retval in self.values():
        retval=random.randint(0,self.N)
      self[key]=retval
      self.inverse[retval]=key
    return retval

def Permute(state,perm,k):
  # perm is the permutation of the number the computational bits form
  # k is a list or tuple of bit indices to apply the gate on
  x=0
  for i,j in enumerate(k):
    if state[0]&(1<<j):
      x+=(1<<i)
  # Bitwise xor to compare input and output bits of the permutation
  y=x^perm[x]
  # Change the bits that should be changed
  for i,j in enumerate(k):
    if y & (1<<i):
      X(state,j)
def InversePermute(state,perm,k):
  try:
    inv=perm.inverse
  except:
    inv=[perm.index(i) for i in xrange(len(perm))]
  Permute(state,inv,k)
```

Appendix B: Python simulation of the QSL Deutsch-Jozsa algorithm

```
#! /usr/bin/env python

# QSL_Deutsch_Jozsa.py
# Version 1.0
# Copyright Jan-Ake Larsson <jan-ake.larsson@liu.se> 2015

from Quantum_Simulation_Logic import *

def Ufbalanced(state,perm):
  Permute(state,perm,xrange(1,state.n))
  CN(state,(state.n-1,0))
  InversePermute(state,perm,xrange(1,state.n))

def Ufconstant0(state,perm):
  Permute(state,perm,xrange(1,state.n))
  pass
  InversePermute(state,perm,xrange(1,state.n))

def Ufconstant1(state,perm):
  Permute(state,perm,xrange(1,state.n))
  X(state,0)
  InversePermute(state,perm,xrange(1,state.n))

### With n=20, printing all function values takes considerable time.
### With n=100, the number of states cannot even be input in xrange().
### Even with n=100000, the below Deutsch-Jozsa algorithm answers in seconds.
n=5
### Choose an oracle: one of UFbalanced, Ufconstant0 or Ufconstant1, and a
### random permutation (can be explicit and non-random if desired, see below)
Uf=Ufbalanced
#Uf=Ufconstant0
#Uf=Ufconstant1
perm=RandPerm(n)
### An explicit permutation can also be used, for example mimicking a
### 4-Toffoli as follows here.
# perm=[0,1,2,3,4,5,6,15,8,9,10,11,12,13,14,7]
### This representation of the permutation is internal to the oracle.  Note
### that the efficiency of the Deutsch-Jozsa algorithm refers to uses of the
### oracle, not the oracle storage or internals. The important feature is that
### the oracle is used only once.

print n,"bit input"
# For small n, the ordinary classical algorithm can be used
if n<16:
  print "All function values"
  for i in xrange(2**n):
    state=QSL_system(i,n)+QSL_system(0,1)
    Uf(state,perm)
    outcome=measureZ(state)
    print "f(%s)=%s"%(outcome[:-1],outcome[-1])

print "Deutsch-Jozsa:",
state=QSL_system(0,n)+QSL_system(1,1)
for i in xrange(state.n):
  H(state,i)
print "(one oracle call only)",
Uf(state,perm)
for i in xrange(1,state.n):
  H(state,i)
outcome=measureZ(state,xrange(1,state.n))
if outcome=="0"*n:
  print "Constant"
else:
  print "Balanced"
```

Appendix C: Python simulation of the QSL Simon's algorithm

```
#! /usr/bin/env python

# QSL_Simons.py
# Version 1.0
# Copyright Jan-Ake Larsson <jan-ake.larsson@liu.se> 2015

from Quantum_Simulation_Logic import *

def Uf(state,s,perm):
  n=state.n/2   # At this point, we have 2n elementary systems
  def Us(state,s): # For each s, construct one common U_s
    prev_i=None
    for i in xrange(n):
      if s[n-1-i]=="1":
        if prev_i!=None:
          CN(state,(prev_i+2*n,i))
          CN(state,(i+2*n,i))
        prev_i=i
      else:
        CN(state,(i+2*n,i))
  state.extend(QSL_system(0,n)) # Add ancilla with computational bits 0
  Us(state,s)                   # U_s acts on input register and ancilla
  Permute(state,perm,xrange(n)) # Permutation on the ancilla
  for i in xrange(n):           # Copy result from ancilla to output register
    CN(state,(i,i+n))
  InversePermute(state,perm,xrange(n)) # Inverse permutation on the ancilla
  Us(state,s)                   # Reset ancilla computational bits to 0
  state.remove(n)               # Remove zeroed ancilla

### Choose a bit mask s, and a random permutation (can be explicit and
### non-random if desired, see below). A long s will work, but it will not be
### possible to print all the function values
#s="00000"
#s="01000"
s="01001"
#s="11111"
#s="".join([random.choice(("0","1")) for i in range(100)]) # 100-bit input
#s="".join([random.choice(("0","1")) for i in range(500)]) # 500-bit input
#s="".join([random.choice(("0","1")) for i in range(1000)]) # 1000-bit input

n=len(s)
### A random permutation
perm=RandPerm(n)
### An explicit permutation can also be used, for example mimicking a
### 4-Toffoli as follows here.
# perm=[0,1,2,3,4,5,6,15,8,9,10,11,12,13,14,7]
### Note that the efficiency of Simon's algorithm refers to uses of the
### oracle, not the oracle storage or internals, here the permutation. The
### important feature is that the oracle is used only linearly many times.

print n,"bit input"
### For small n, the function values can be compared
if n<10:
  print "All function values"
  for i in xrange(2**n):
    state=QSL_system(i,n)+QSL_system(0,n)
    Uf(state,s,perm)
    outcome=measureZ(state) # All bit indices
    print "f(%s)=%s"%(outcome[0:n],outcome[n:2*n])
  print

##########################################################################

print "Simon's algorithm:"
outcomes=[]
basis=[]
while len(basis) < n-1:
  ### Simon's subroutine
  state=QSL_system(0,2*n)
  for i in xrange(n,2*n):
    H(state,i)
  Uf(state,s,perm)
  for i in xrange(n,2*n):
    H(state,i)
  y=measureZ(state,xrange(n,2*n))
  ### End Simon's subroutine
  outcomes.append(y)
  # remove linear span of outcomes from y
  for v in basis:
    k=v.index("1") # returns first index of a set bit
    if y[k]=="1":
      y+=v
  # Add y to the basis if y is still nonzero
  if y!="0"*n:
    # Remove the component along y from previous basis vectors
    k=y.index("1") # returns first index of a set bit
    for j,v in enumerate(basis):
      if v[k]=="1":
        basis[j]+=y
    basis.append(y)
if n<10:
  print "Outcomes:", outcomes
if n<10:
  print "Outcome basis:", basis
### s-star is orthogonal to the set of outcomes
sstar="1"*n
for v in basis:
  if v.dot(sstar):
    sstar+=v
### Test the proposed s-star: is f(0)==f(s-star)?
state=QSL_system(0,2*n)
Uf(state,s,perm)
outcome0=measureZ(state,xrange(0,n))
state=QSL_system(sstar,n)+QSL_system(0,n)
Uf(state,s,perm)
outcomesstar=measureZ(state,xrange(0,n))
if outcome0!=outcomesstar:
  sstar="0"*n
###
print "After", len(outcomes),"oracle calls, computed s:",sstar

#############################################################################

print "Modified BH algorithm,", n, "oracle calls:"
outcomes=[]
for k in xrange(n):
  ### Modified BH subroutine
  state=QSL_system(0,n)+QSL_system(1<<(n-1-k),n)
  for i in xrange(2*n):
    H(state,i)
  Uf(state,s,perm)
  for i in xrange(n,2*n):
    H(state,i)
  outcomes.append(measureZ(state,xrange(n,2*n)))
  ### End modified BH subroutine
if n<10:
  print "Outcomes:", outcomes
### s is orthogonal to the set of outcomes
sstar="1"*n
for v in outcomes:
  if v.dot(sstar):
    sstar+=v
print "Computed s:",sstar
```